Deriving Software Usage Patterns from Log Files

Mark Guzdial
GVU Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Mark.Guzdial@cc.gatech.edu

Abstract

Log files (discrete recordings of user actions during software use) offer the ability to collect human-computer interaction data on a number of users, over time, while the users are engaged in typical tasks in typical environments. The disadvantage of log files is the lack of automated methods for analyzing the volumes of data in a meaningful way. This paper presents a log file analysis tool, Hawk, and discusses the characteristics which make it useful for this task. A particular analysis technique, based on Markov chain analysis, is described which can be used to derive high-level software usage patterns. A study of student interactions with a programming environment are used for examples of the use of the the tool and the technique.

Log files (that is, discrete recording of user actions during software use) have several characteristics which make them ideal for research on the design of user interfaces and on the interactions between humans and computers. They can be used to collect data on any number of users over time, during each and every use of the software. Particularly important is that log files can be used to collect data while users are working on typical tasks in typical environments. More traditional methods for gathering data on human-computer interactions, such as think-aloud protocols, require unusual settings that can confound the analyses.

The disadvantage of log file data is that there are few analysis techniques. Log file data tends to be voluminous, and often at too low a level to be of much use without some aggregation (e.g., individual keystrokes), suggesting the need for automated analyses. Researchers in hypertext and hypermedia have developed techniques and tools for using log files to trace access and then to develop an assessment of student concept formation in terms of information accessed (e.g., Horney and Anderson-Inman, 1992; Dershimer, Berger, and Jackson, 1991) and to characterize student navigation styles in terms of information accessed (Hutchings, Hall, and Colbourn, 1992).

Log file analysis techniques for exploring user interactions with other forms of software are more rare. Hammer and Rouse (1979) used Markov chain analysis (a standard analysis tool, e.g., Kemeny, Snell, & Thompson, 1974) to study keystrokes in editing text files, but found the technique to be too sensitive to individual differences. Winne and Gupta (1993) have identified a number of powerful measures to use in evaluating log files of students using a computer-based study aid. The techniques of Winne and Gupta are based on data and graph theory, and they

include forms of Markov analysis. However, they have not yet presented tools for performing these analyses nor the results of applying these techniques to evaluation of real student data

I have developed a log file analysis tool (*Hawk*) for use in studying student use of an end-user programming environment, *Emile*. One of the techniques I have developed for studying student use is to describe usage patterns in terms of a Markov chain. This characterization allows me to determine where the critical points were in the interface and what was the user's general flow in their use of that interface. This paper describes:

- Emile, with special emphasis on the format of the log files;
- Hawk, with discussion of what makes a useful log file analysis tool;
- And the technique, presented in a general form.

The summary section comments on other applications of this method and on future directions.

I. Emile

Emile is a design support environment for high school students programming in multimedia to teach other students about physics. Students using Emile during a summer workshop created animated simulations and demonstrations of kinematics (with video, sound, graphics, and speech), while at the same time, improving their own understanding of physics and programming. Emile, its use, and the learning of students using Emile are my dissertation work (Guzdial, 1993.)

Emile (Figure 1) is built upon Apple's HyperCard programming language for Macintosh microcomputers. Emile extends HyperCard with explicit support to aid users (especially novices) in dealing with the complexity of design, both in terms of process and product. Emile provides multiple representations of the program being designed, prompts for reflection at various stages in the design process (e.g., Predictions, Journals), meta-level design components for structuring the artifact (Goals and Groups), process structuring to encourage top-down design, a component library to bootstrap design projects, and high-level user preferences which allow users to tailor the support structures to their individual needs.

Emile is structured around a Design Notebook and a Project Window (Figure 1). The Design Notebook has various kinds of pages, including pages for reflection (e.g., Journal pages), pages containing representations (e.g., the graphical Project Chart and the textual Table of Contents), and pages that describe components (e.g., Button pages.) All components described in the Design Notebook appear and are manipulable in the Project Notebook. The user has a number of navigation methods available for moving between pages in the Notebook.

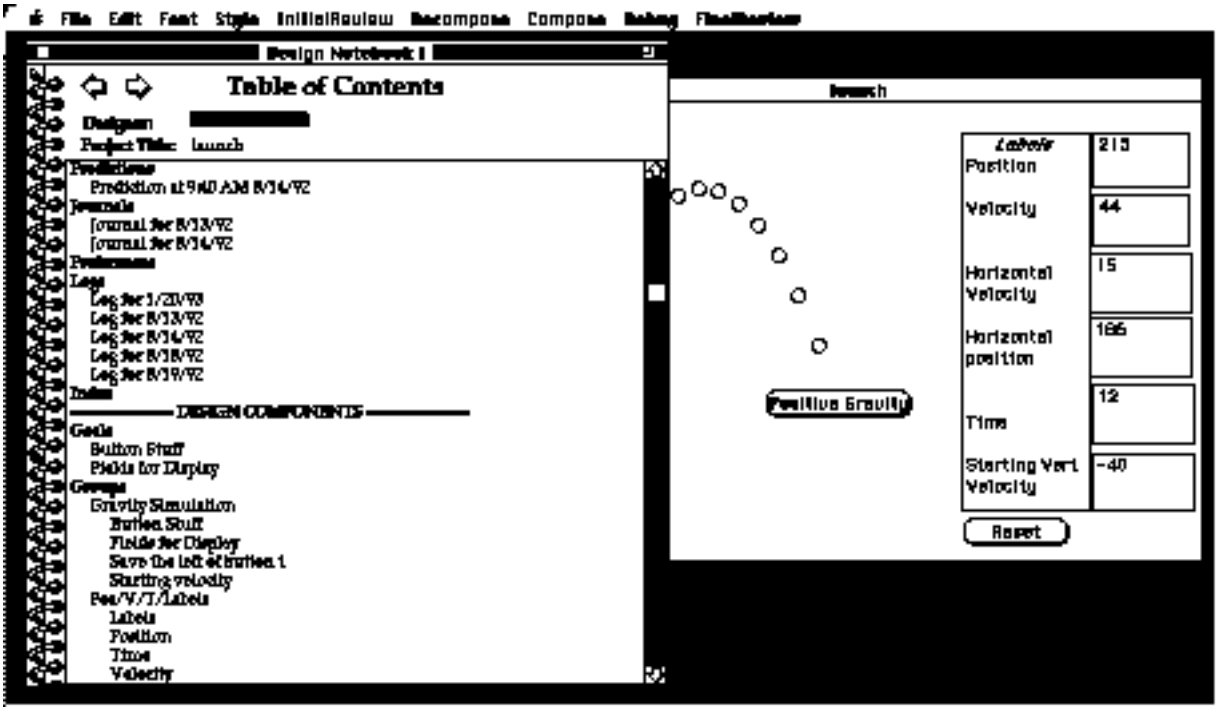

Figure 1: Emile, A Supportive Environment for Science Learners Programming in Multiple Media

A key question in my study is how students used Emile. I wanted to know if there were differences in software use that corresponded to physics learning differences (as measured by clinical interviews before and after the workshop) or to differences in the kind and complexity of programs the students completed. The goal was not to determine causality (e.g., use of tool X led to better learning), but to determine if there were differences, and if so, where those differences might lie.

To address this question, I modified Emile to create process trace or log files. Every user mouse click, menu operation, and text entry was time stamped and logged. Keystroke level data was not recorded. A typical log file for a day ranged in size between 20 and 30K.

Each entry had an identical format: a single line, with elements separated by commas, identifying the date, time, general type of page, specific name of page, the activity the user was engaged in, and any detail on that activity. Some example log file entries are:

```
date,time,page name,page type,activity,detail
8/13/92,10:08:25 AM,Project Description,Chapter Heading,Edited,Project
Description
8/13/92,10:16:20 AM,A Droppable Object,Button,menuSelect,copyToNotebook
```

The first entry indicates that the user edited the Project Description on the Project Description page, and the second indicates that the user made a menu selection to copy the button "A Droppable Object" to his notebook from the component library.

Emile's log file format is redundant, e.g., the time stamp need not be available on every entry, and the page name and type does not change on every user action. I chose a redundant format for Emile for two reasons.

1. A redundant format provides some insurance against missing information in the log file. On occasion, I have found user actions that were not properly logged. I have been able to infer what action and at what approximate time the operation took place by inference from the surrounding log entries (e.g., a change in position from an unlogged navigation operation, a change in name or context from an unlogged editing operation.)
2. In conducting log file analysis on another programming environment (Guzdial et al., 1992), I found that a non-redundant log file format required the analysis program to store a significant amount of context information while processing the log. Analysis of the redundant format is made easier since much of the context information is available at each entry.

II. Hawk

The characteristics that I wanted for a log file analysis tool were that it should be flexible enough that I could explore a variety of analysis techniques and that it should be easy to specify simple analyses (that is, filtering, counting, and recoding.) I decided that I wanted a programming language at the core of the tool, to enable the flexibility I desired, and that I wanted that language embedded in an environment which would aid the analyses.

I based Hawk on the UNIX programming language Awk (Aho, Kernighan, & Weinberger, 1988) Hawk is (HyperCard AWK) is an interpreter and pseudo-compiler for a variant of Awk embedded in a HyperCard stack. The language provides the processing power for analyzing the log files. The surrounding stack creates an environment for managing data files (log files, recoded data files, and final results), storing and organizing analysis programs, and generating batch runs. The result is a flexible, powerful tool for conducting log file analysis.

A. The Hawk Language

I modified the Awk programming language to specialize it for log file analysis in HyperCard. This section briefly describes Hawk as a log file analysis language.

A Hawk program consists of a number of pattern-action pairs. Each line of the input data file is sequentially compared to the pattern of each pair. If the pattern matches, the action is executed. There are special patterns which match true before any input is read (**BEGIN**), after all input is read (**END)**, and for every line (**EVERY**).

Variables are defined by use, without type declarations, and their initial values are always zero or empty (depending on context.) Special variables are used to access the input data line. The entire line is referred to as **$0**. Each field (or item) on the input line is accessed numerically: **$1** is the first field, **$2** is the second, and so on. The field delimiter defaults to a comma, but can be set to any character. Various relational operators are used with special variables to test the input line, such as **~** for contains, **!~** for does not contain, and the standard relational operators =, <, and >. Standard real number arithmetic is allowed in either patterns or actions.

Filtering is achieved by specifying patterns that match only the desired data and printing. For example, a program that filters out entries that do not refer to the Project Chart (Emile's graphical representation of the developing design) looks like this:

```
$0 ~ "Project Chart" {print $0}
```

Recoding is a matter of changing what's printed: Instead of printing the original input, a Hawk program for recoding would print the recode of the input line. An example of recoding will be seen in the next section.

Counting events in Hawk is made simple by using a powerful feature of Awk that Hawk inherited: *associative arrays*, which index by any word or phrase, as well as by number. For example, a program that counts the number of times that each kind of page was visited in Emile is only some six lines long. In the Emile log format, the fourth field indicated the kind of page the user visited. The following short program counts visits to page types.

```
$4 != lastVisited
   {count[$4] := count[$4] + 1;
    lastVisited := $4}
END
   {for (i in count)
        {print i,count[i]}}
```

This program remembers the last page type visited (in **lastVisited**, whose initial value is empty or null) to prevent multiple logged actions from appearing as new visits to the page type. The **count** array is incremented for each new visit to a page type (**$4**). After all the input is processed, the pattern **END** matches, and the array **count** is output by walking it using a **for** loop with an index variable **i**.

Hawk added a few new functions to Awk in order to facilitate log file processing. Two important examples are:

- The function **seconds()** function converts from any HyperCard recognized date and time format to the number of seconds since midnight, January 1, 1904. The **seconds()** function is a tool for using the time stamps in the log file to compute time intervals, such as the amount of time spent in a particular operation.
- The function **value()** takes a string which evaluates to any HyperCard-valid expression, evaluates that expression, and returns its value. This function makes available the full range of HyperCard numeric and text processing to a Hawk program.

Hawk also changed several of the rules of pattern-action pairs in Awk. The most important is that a Hawk program can contain multiple **BEGIN** and **END** patterns, while Awk allowed only one each per program. The advantage of multiple **BEGIN** and **END** patterns is that programs can be combined by literally concatenating the pattern-action pairs of one program to the pattern-action pairs of another, without any rewriting. This feature is useful when combining analyses. For example, I combined a program to output the types of cards visited with a program to output the kind of notebook navigation used. By literally concatenating one program to another, I was able to get a new report which interleaved the type of page visited with the kind of navigation used to move between pages.

Hawk is a powerful language for log file analysis because simple analyses are easy to specify.. Its basis in pattern-action pairs allow for flexible combination of programs, and its functionality provides support for a range of log file analysis functionality: filtering, recoding, and counting.

B. The Hawk Tools Stack

Hawk programs are combined in a Hawk Tools stack. A HyperCard stack contains a number of *cards* organized in a linear order in a *stack*. Only one card can be visible at a time. Figure 2 shows a typical program card in the stack.

Each program card in the stack contains a single Hawk analysis program. The box containing text (called a *field* in HyperCard) at the top of the large window is for naming the card (the first line of the field, by convention) and for commenting on it. The small window at the right lists all the card names in this stack. It serves as a table of contents and navigation mechanism: clicking on any name in the list shows that card in the main window. The arrows in the upper left hand corner of the main window allow for movement linearly among cards. Standard HyperCard functions can also be used for reordering cards or navigating among cards.

Other cards in the stack provide generic functions. For example, the Batch Card at the front of the stack takes a list of cards for batch compilations or executions.

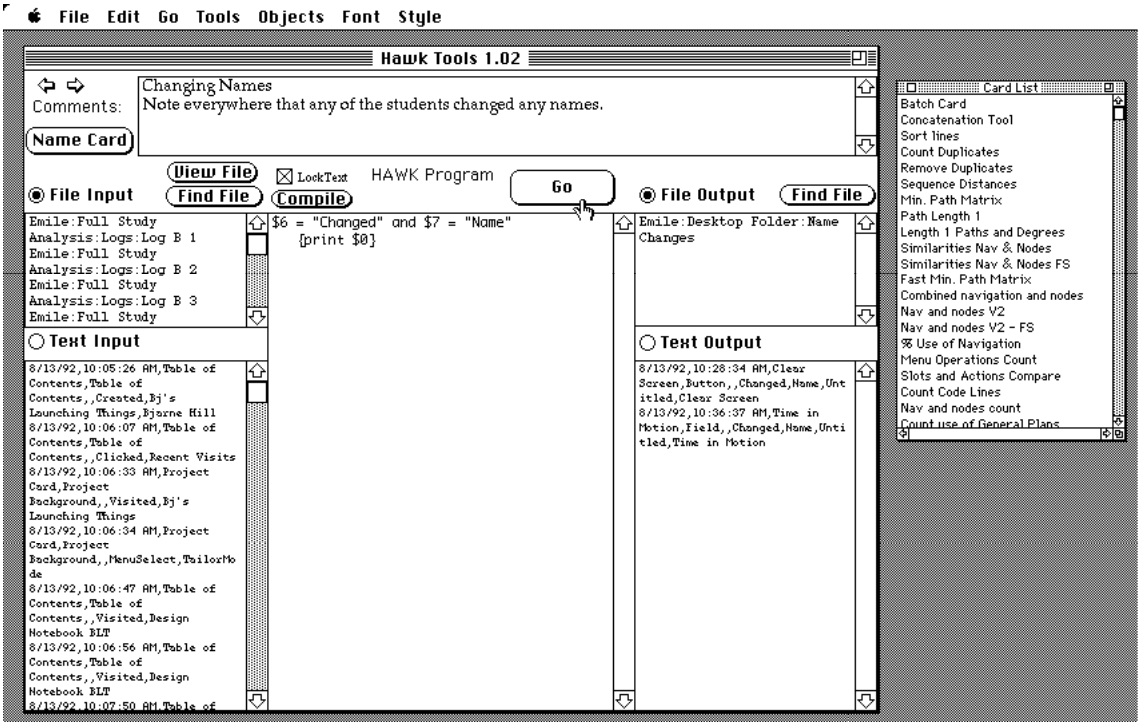

Figure 2: Program Card in the Hawk Tools Stack

A program card's functionality is contained in its three columns: input, program, and output. Input (the leftmost column) can come from files (upper field) or text appearing on the card (lower field). Output (rightmost column) can go either to files (upper field) or on–card text (lower field.) Radio buttons select between file and text options.

Hawk is flexible in its input and output specification. Operations are provided for viewing input files for selection, or finding files for input or output. Input and output can be represented by fields right on the card. Input files can be specified by full path name (in the Macintosh's hierarchical filing system) or by folder (a Macintosh file directory) path name. Output can be directed to one

file (that is, the results of the analysis on each input file is concatenated into the one output file) or into multiple files.

The center column is where the Hawk program is entered. Both a Hawk language interpreter and pseudo-compiler are provided. The Hawk interpreter is functional but slow. The Hawk compiler compiles Hawk program to HyperCard programs, which execute up to three times faster. In actual use, the interpreter is often used during debugging a new analysis program on a sample of the input data, then the compiler is used to create a faster analysis for use on the entire set of data.

The Hawk Tools stack provides a visually well-organized environment for developing and conducting analysis. By collecting all the analysis programs and providing navigation access between them, the stack serves a generative function. Creating a new analysis is often a matter of copying and tailoring pieces out of other analysis programs instead of writing a new Hawk program.

III. Analyzing Log Files using Markov Chains

Graphical user interfaces provide a method of segmenting the kinds of actions users undertake into low-level actions (e.g., keystrokes) and a higher-level actions (e.g., menu selections). I took advantage of this segmentation to define high-level actions which defined *stages* of Emile use. Deriving a pattern of student use then becomes a matter of transitions between stages, that is, analyzing the log file as a Markov chain. Additionally, further analyses can be undertaken in terms of these stages, e.g., noting what low-level actions occur during each stage.

Markov analysis refers to transitions between states. The log file technique discussed here has a five step process for using states to describe user stages:

1. **Definition and Classification:** Definition of user states/stages and classification of log entries into those states.
2. **Computing Probabilities:** Computing the observation probability of transitions from one state/stage to another.
3. **Drawing Transition Diagrams:** The resultant transition matrix can be used to draw transition diagrams which describe the flow between states.
4. **Computing Steady State Vectors:** The steady state vector reflects the overall probabilities of a user being in any one state.
5. **Computing Expected Probabilities:** An expected steady state vector can be created which can be used to compare to the observed steady state vector.

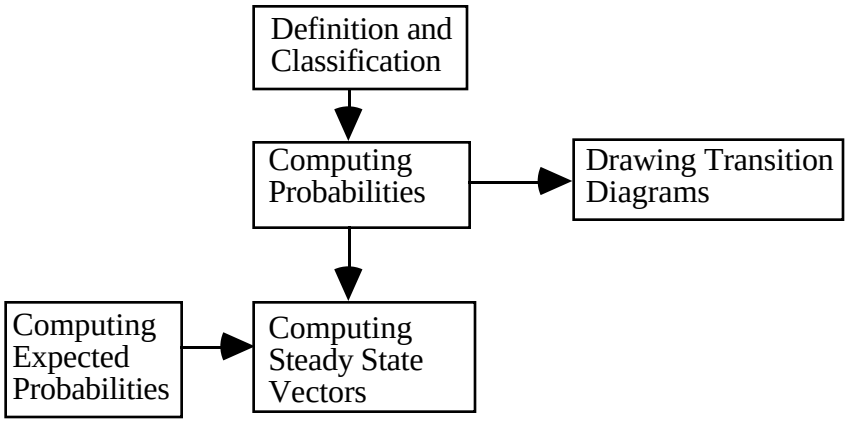

Figure 3: Process Flow of Markov Log File Analysis

The rest of this section presents each of these steps in detail using examples from the analysis of Emile.

Definition and Classification: Emile was developed to provide support for students engaged in design. The stages used in studying Emile usage grew out of research in design and problem-solving (e.g., Schoenfeld, 1980.) A description of the use of Emile in terms of these design stages would provide insight into the significant stages during actual use and how the user's design style with Emile changed over time. The design stages defined for Emile were:

- **Initial Review**: Reviewing the task and making plans. (By definition, the student begins each day in Initial Review.)
- **Decomposition:** Defining the components of the solution.
- **Composition:** Assembling the components into a whole.
- **Debugging:** Testing the whole solution.
- **Final Review:** Review work completed and considering what might be useful to reuse in future problems.

Classifying the user's operations into design stages was particularly easy because the menu operations were already classified into the five stages on the menu bar. The five menus in Emile were named after the five stages in order to facilitate the students' metacognitive reasoning about design by giving them language for the kinds of activities in which they were engaged. (For more on the educational rationale behind this technique, see Farnham-Diggory, 1990 and Paris, Wasik, and Turner, 1989.) For example, creating a Plan page or a Project Description was classified as Initial Review activities; creating a new component was a Decomposition activity; adding a component to the Project Window was a Composition activity; running the program was a Debugging activity; and creating a Journal entry was a Final Review activity.

A Hawk program was written to recode a log file into a sequence of design stage markers, part of which is seen below.

```
$6 = "MenuSelect" and
$7 = "Choose Project"
  {print "********","Initial Review"}
$6 = "MenuSelect" and
$7 = "Make Plans"
  {print "********","Initial Review"}
$6 = "MenuSelect" and
$7 ~ "New"
  {print "********","Decomposition"}
$6 = "Did" and
$7 ~ "compose"
  {print "********","Composition"}
$6 = "MenuSelect" and
$7 = "Test"
  {print "********","Debug"}
$6 = "MenuSelect" and
$7 = "MakeJournal"
  {print "********","Final Review"}
```

By marking the design stages with asterisks, I could interleave low-level information between the design stage markers and write additional Hawk programs to analyze low-level user actions in terms of higher-level events. For example, by interleaving the kind of pages visited, I could determine which page types were most often visited between which design stages.

Computing Probabilities: Another Hawk program read in the design stage recoding and calculated the transitions between design stages. This program computed the observed probability

of entering stage B from A by counting the number of transitions from A to B over the number of times that stage A was left[1].

The result of computing probabilities is a transition matrix describing the probability of moving from one stage to another. Table 1 presents a sample transition matrix, one derived from the log file of student M working on his fourth and last program in the workshop. One reads this matrix to say that the probability of an Initial Review operation following another Initial Review operation for this student for this program was 25%. The transition from Initial Review to Decomposition also occurred 25% of the time. A transition from Initial Review to Composition never occurred for this student on this program.

	Initial Review	**Decomposition**	**Composition**	**Debugging**	**Final Review**
Initial Review	0.25	0.25	0.00	0.50	0.00
Decomposition	0.00	0.63	0.27	0.10	0.00
Composition	0.00	0.10	0.52	0.38	0.00
Debugging	0.02	0.01	0.07	0.89	0.01
Final Review	0.50	0.50	0.00	0.00	0.00

Table 1: Transition Matrix for Student M, Program 4

Drawing Transition Diagrams: Given the transition matrix, a transition diagram can be drawn describing the general flow of user interaction. While not absolutely necessary, such a diagram is a powerful representation for determining trends and patterns in a student's use of Emile. Figure 3 is the transition diagram for the matrix in Table 1. The diagram allows us to pick out interesting relationships in the user's design style with Emile on this program. For example, the student more often transitioned from Debugging to Composition (52% of all transitions out of Debugging) than to Decomposition (10%). This suggests that the student found most often that problems in the program were due to how things were put together rather than how things were defined.

[1]Strictly speaking, the probabilities should more accurately have been computed with the denominator as the number of times that stage A was entered. But in this form, the last stage the user was in when they ended the program becomes a *dead* state, in Markov analysis terms, which makes the analysis more difficult. The form used here does not seriously change the results of the analysis

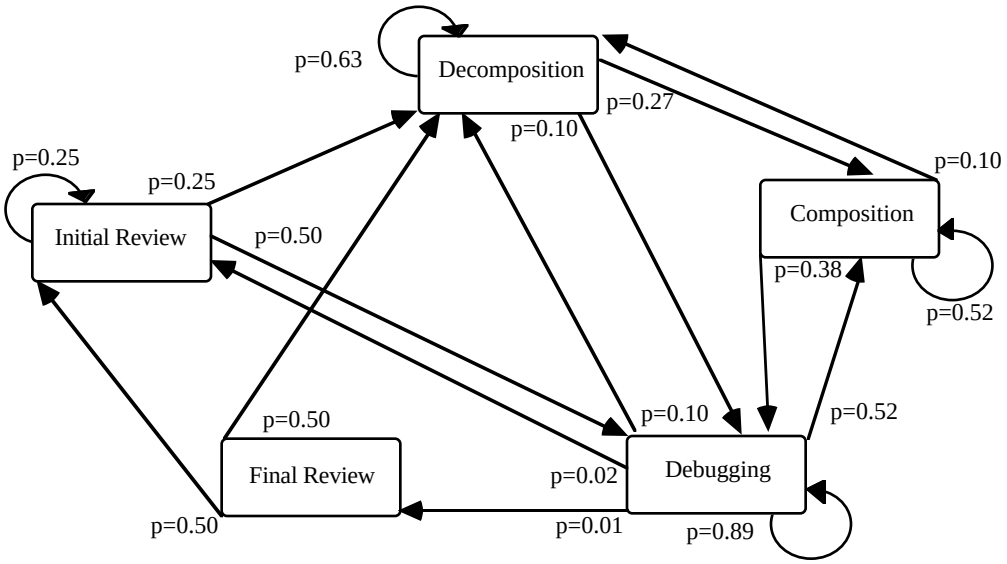

Figure 3: Transition Diagram for Student M, Program 4 (Probabilities are marked at the head of a transition arrow.)

Compare the transition diagram in Figure 3 to the transition diagram for student C in Figure 4. Both of these diagrams are representing students at the same level of experience with Emile and the same point in the workshop, though not exactly the same program. (Program 4 was a student-selected project.)

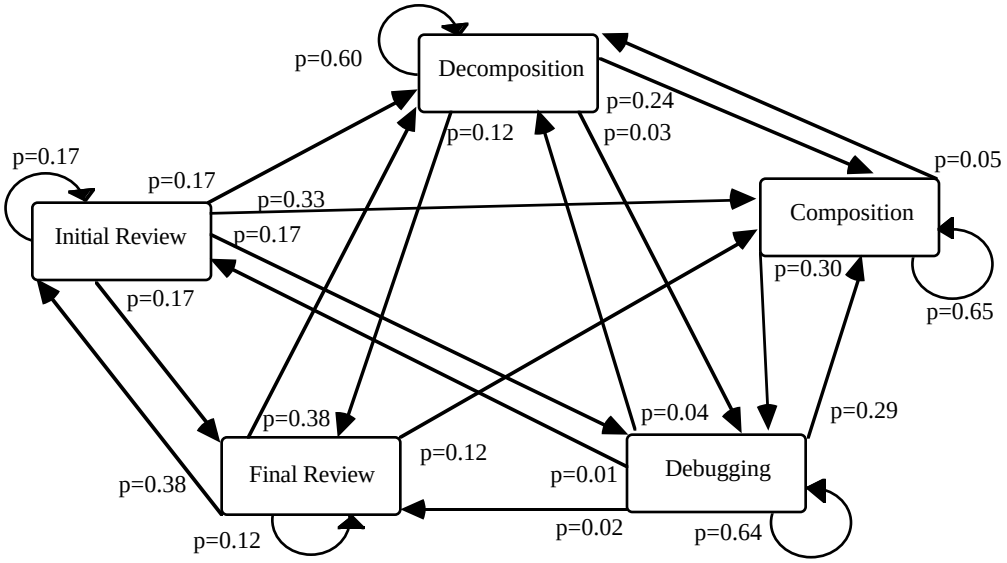

Figure 4: Transition Diagram for Student C, Program 4

Note that Figure 4 has several more transitions than does Figure 3 (16 versus 21) which implies that Figure 4 has lower probabilities than Figure 3. These suggest that student C moved more often between design stages than student M and that student C moved to more different stages from any one stage than student M.

One might hypothesize that the additional transitions suggest a more flexible approach. I suggest instead that student C "thrashed" more in his design process – that he was inexperienced and didn't have good heuristics for how to design. Other data collected in the study of Emile indicate that student M was a more successful programmer: He had previous programming experience, and the programs he created during the class were more complex and more successful than student C's. Student C might have been well served by more process support in Emile to guide him though a successful design process.

Computing Steady State Vectors: A transition matrix can be solved to come up with a vector reflecting the probability that any one operation chosen from random from the log file will belong to a particular stage. The steady state vector is a measure of the relative number of operations in any one state. The steady state vector can be computed by solving the transition matrix or by simply multiplying the matrix by itself until each column holds the same value in all row vectors, to the desired accuracy. I used the latter method – repeatedly multiplying the matrix by itself (and the result by itself) until the columns agreed to three decimal positions.

Student	Initial Review	Decomposition	Composition	Debugging	Final Review
B	0.04	0.14	0.52	0.29	0.02
C	0.05	0.14	0.39	0.38	0.02
L	0.06	0.11	0.48	0.31	0.05
M	0.04	0.12	0.27	0.56	0.01
S	0.06	0.13	0.29	0.44	0.02

Table 2: Average Steady State Vectors for Students using Emile

Table 2 presents the average steady state vector for the five students who took the summer workshop using Emile. Each of these vectors is the average of the steady state vectors for each student on each of the four programs completed during the workshop. For example, the entry for student B under Decomposition in Table 2 says that the probability of any random operation of B's being a Decomposition operation was 14%, or, in other words, that 14% of all of B's operations were Decomposition operations.

One way to interpret the steady state vector is in contrast with similar environments. Most of the student's activity occurred in the Composition and Debugging design stages. This is in marked contrast to work on another student programming environment, the GPCeditor, in which 30% of the student's activity occurred in the Decomposition design stage (Soloway, et al., 1993). The results in Table 2 suggest that Emile students do less work in Decomposition than in Composition and Debugging. Detailed interpretation requires evaluation of what use of each environment required in terms of these stages (e.g., Emile may not require as many Decomposition operations in design as did the GPCeditor) and what each operation did (e.g., Emile's operations may combine several GPCeditor operations.)

The steady state vector can suggest where future software design effort should be directed. For two students (M and S), Debugging was by far the most common kind of operation that they engaged in, at an almost 2:1 ratio with the stage with the next highest probability. For the other three students, Debugging is the second most common state after Composition. The high probability of Debugging operations says that the students spent a lot of effort doing Debugging, and that future versions of Emile might better support students by providing more Debugging

support. Since the students using Emile are creating Physics simulations, this Debugging support might look like the graphing and visualization tools used by computational scientists who base their work in computer simulations.

Computing Expected Probabilities: In order to consider the results of Table 2 as reflecting user interaction with the software rather than simply software design, an expected steady state vector should be computed. I chose as my base assumption that use could be understood in terms of the overall percentage of available operations. The fact that all the Decomposition probabilities are between 0.11 and 0.14 could be explained if 11 to 14% of all operations available in Emile were Decomposition operations – we would expect these results if all operations were exercised equally, with no preference suggested by the usability of the various operations and representations in the software. In Emile, the actual percentage of Decomposition operations compared to the whole is 28%. (See Table 3 for all the percentages.) We can safely assume, then, that the steady state vectors suggest a user preference for particular operations during use of the software.

Initial Review	Decomposition	Composition	Debugging	Final Review
0.08	0.28	0.44	0.10	0.10

Table 3: Percentage of Total Operations in Each Design Stage for Emile

IV. Summary and Extensions

The key points of the previous three sections are:

- A redundant log file format can be useful in providing insurance against missing log entries and in making analysis easier with additional context information.
- Log file analysis can be seen as a form of text processing, and a text processing language, tuned for log file analysis, can be a powerful tool, permitting flexible analysis in brief expressions.
- Embedding that language in an environment that helps in reuse of previous analyses is a tremendous benefit.
- Finally, significant leverage on log file analysis can result from identifying high-level actions and viewing software usage in terms of transitions between these actions.

The technique described in this paper has proven useful in describing patterns in student use of Emile. Besides evaluating design process, similar analyses have been conducted to note the kinds of tools used (by defining each tool a different state) and the level of focus in the design (e.g., how much effort are students expending on statement-level programming versus high-level module structuring.)

Hawk has also proven useful in analyzing other data, cast in a log-like format. *Coding Book* is a tool for qualitative coding of data that was developed to analyze the student interviews in the Emile study. Coding Book features a flexible report mechanism that can output data on cases and codes in a format like Emile's log files. Hawk can then be used to test hypotheses and summarize the results of a coding effort.

Current work explores the use of Hawk for direct comparisons between sets of log file data. Hawk as presented here analyzes each log file individually. The newer versions of Hawk extend this functionality with global variables (for comparing across input files) and measures of difference, such as sequence distance.

Acknowledgments

My advisor, Elliot Soloway, encouraged the focus on log file analysis in my study and provided resources for the work. Discussions with Pat Baggett, Carl Berger, Charles Dershimer, Andrzej Ehrenfeucht, Wendy Hall, Gerard Hutchins, and David Jackson were helpful in exploring log file analysis techniques. Michael Konneman and Chris Walton helped evaluate Hawk for log file analysis in their study of log files for GPCeditor and SODA, two Pascal-based programming environments. This research was supported in part by National Science Foundation grant #MDR-9010362 and by Apple Computer, Inc.

References

Aho, A.V., B.W. Kernighan, & P.J. Weinberger. (1988) *The AWK Programming Language.* Reading, MA: Addison-Wesley.

Dershimer, C, C. Berger, & D. Jackson. (1991) *Designing Hypermedia for Concept Development: Formative Evaluation through Analysis of Log Files.* Paper presented at the National Association for Research in Science Teaching annual meeting, Fontana, WI.

Ericsson, K.A. & H.A. Simon (1984) *Protocol Analysis*. Cambridge, MA: MIT Press.

Farnham-Diggory, S. (1990) *Schooling.* Cambridge, MA: Harvard University Press,.

Guzdial, M., E. Soloway, P. Blumenfeld, L. Hohmann, K. Ewing, I. Tabak, K. Brade, and Y. Kafai. (1992) "The future of CAD: Technological support for kids building artifacts." In *Learning to Design, Designing to Learn: Using Technology to Transform the Curriculum*. D. Balestri, S. Ehrmann, and D.L. Ferguson (Eds.) Ablex: Norwood, New Jersey.

Guzdial, M. (1993) *Emile: Technological Scaffolding for Science Learners Programming in Multiple Media.* Ph.D Thesis at the University of Michigan. In Preparation.

Hammer, J.M. and W.B. Rouse. (1979) Analysis and modeling of freeform text editing behavior. In *Proceedings fo the 1979 International Conference on Cybernetics and Society*, Denver, CO.

Horney, M.A., L. Anderson-Inman. (1992) *The ElectroText Project: Hypertext Reading Patterns of Middle School Students.* Paper presented at the annual conference of the American Educational Research Association, San Francisco, CA.

Hutchings, G.A., W. Hall, & C.J. Colbourn. (1992) *A Quantitative Study of Students' Interactions with a Hypermedia System.* Technical Report CSTR 92-04, University of Southampton, Department of Electronics and Computer Science, UK.

Kemeny, J.G., J.L. Snell, & G.L. Thompson. (1974) *Introduction to Finite Mathematics.* Englewood Cliffs, NJ: Prentice-Hall.

Paris, S.G., Wasik, B.A., and Turner, J. 1989. The development of strategic readers. *Handbook of Reading Research*. Pearson, P.D. (Ed.) New York: Longman.

Schoenfeld, A.H. (1980). Teaching problem-solving skills. *Amerian Mathematical Monthly . 87.* 794-805.

Soloway, E., M. Konemann, C. Walton, M. Guzdial, & L. Hohmann. (1993) *The GoalPlanCode Editor: What Did the Students Learn?* In preparation.

Winne, P.H. and L. Gupta. (1993) *Data and Graph Theory Measures for Modeling Cognitive Strategies in Tutorials with STUDY.* Paper presented at the annual meeting of the American Educational Research Association, Atlanta, GA.